Introduction to MPI Profiling with Allinea MAP

Welcome! These notes are a companion to a hands-on course run by Allinea staff, which comes with full example code and MAP trace files. If you haven't attended this course, take a look at http://www.allinea.com/help-and-resources/training/ and contact Allinea to book one. They're really good!

Session 2: CPU Optimization

As in session 1, we have a simple example program that is attempting to generate some random numbers, sqrt them all then find the maximum of those square roots. It currently looks like this:

```
for(j = 0; j < ITERATIONS; j++){
    /* split the array into sections and send these to workers */
    if(myid == 0){
        subsize = (int)floor(ARRAY_SIZE/numprocs);
        sIndex = subsize + ARRAY_SIZE%numprocs;

        /* send the size of each section to the workers */
        for(i = 1; i < numprocs; i++){
            MPI_Send(&subsize, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
            MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
        }
        for(i = 1; i < numprocs; i++){
            MPI_Recv(&result, 1, MPI_DOUBLE,  i, 0, MPI_COMM_WORLD, &status);
            maxes[i] = result;
        }
        max = maxes[1];
        for(i = 2; i < numprocs; i++){
            if(max < maxes[i]){
                max = maxes[i];
            }
        }
        results[j] = max;
    }
    else{ /* receive a subsection of the array and find the max in it */
        MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

        /* fill the input data with random numbers ourselves */
        srand(time(NULL));
        for(i = 0; i<subsize; i++)
            numbers[i] = rand();

        for(i = 0; i<subsize; i++)
            numbers[i] = sqrt(numbers[i]);
        result = numbers[0];
        for(i = 1; i<subsize; i++){
            if(result < numbers[i])
                result = numbers[i];
        }
        MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
}

//Calculates the average, removes the largest and smallest from all the runs
if(myid == 0){
    max = results[0];
    for(i = 0; i<ITERATIONS; i++){
        if(max < results[i])
            max = results[i];
    }
    printf("Max after %d iterations:%g\n", ITERATIONS, max);
}
```

Note: if you look at the code more closely, you'll see this is an obviously flawed algorithm - a better solution would be to find the max of the random numbers and ***then*** *take its square root, saving a lot of computation! This is a prime example why step-by-step optimization should* ***always*** *be combined with thinking about what the algorithm is trying to achieve. For the sake of this example, though, we're going to assume it's important to perform the square root on each number before finding the largest of them, which helps us mark the distinction between improving the* ***algorithm*** *and improving its* ***implementation****.*

This is the code we ended up with after session 1. This time we're going to improve it further. We'll start by compiling and running the code to get a baseline timing:

```
mark@Namaste-II ~/2-cpu-optimization/problem $ mpicc -g -O3 sqrtmax-workergen.c -o
sqrtmax-workergen

mark@Namaste-II ~/2-cpu-optimization/problem $ time mpirun -n 4 ./sqrtmax-
workergen
Max after 500 iterations:46340.9

real    0m3.620s
user    0m10.373s
sys     0m0.024s
```

Good, now we'll run MAP on it and analyze the output. Instead of using interactive sessions, you can also run MAP in batch mode with the -profile argument. We'll do that this time:

```
mark@Namaste-II ~/2-cpu-optimization/problem $ map -profile -n 4 ./sqrtmax-
workergen
Allinea MAP map-dev, (c) Allinea Software Ltd
Profiling                    : /home/mark/Work/code/doc/training/map-
introduction/2-cpu-optimization/problem/sqrtmax-workergen
Processes                    : 4
MPI                          : OpenMPI
Statically linked MPI wrapper: no
Static linked sampler        : no

MAP starting profiling...
mpirun: Max after 500 iterations:46340.9

MAP analysing program...
MAP gathering samples...
MAP generated sqrtmax-workergen_4p_2013-03-11_11-40.map
```

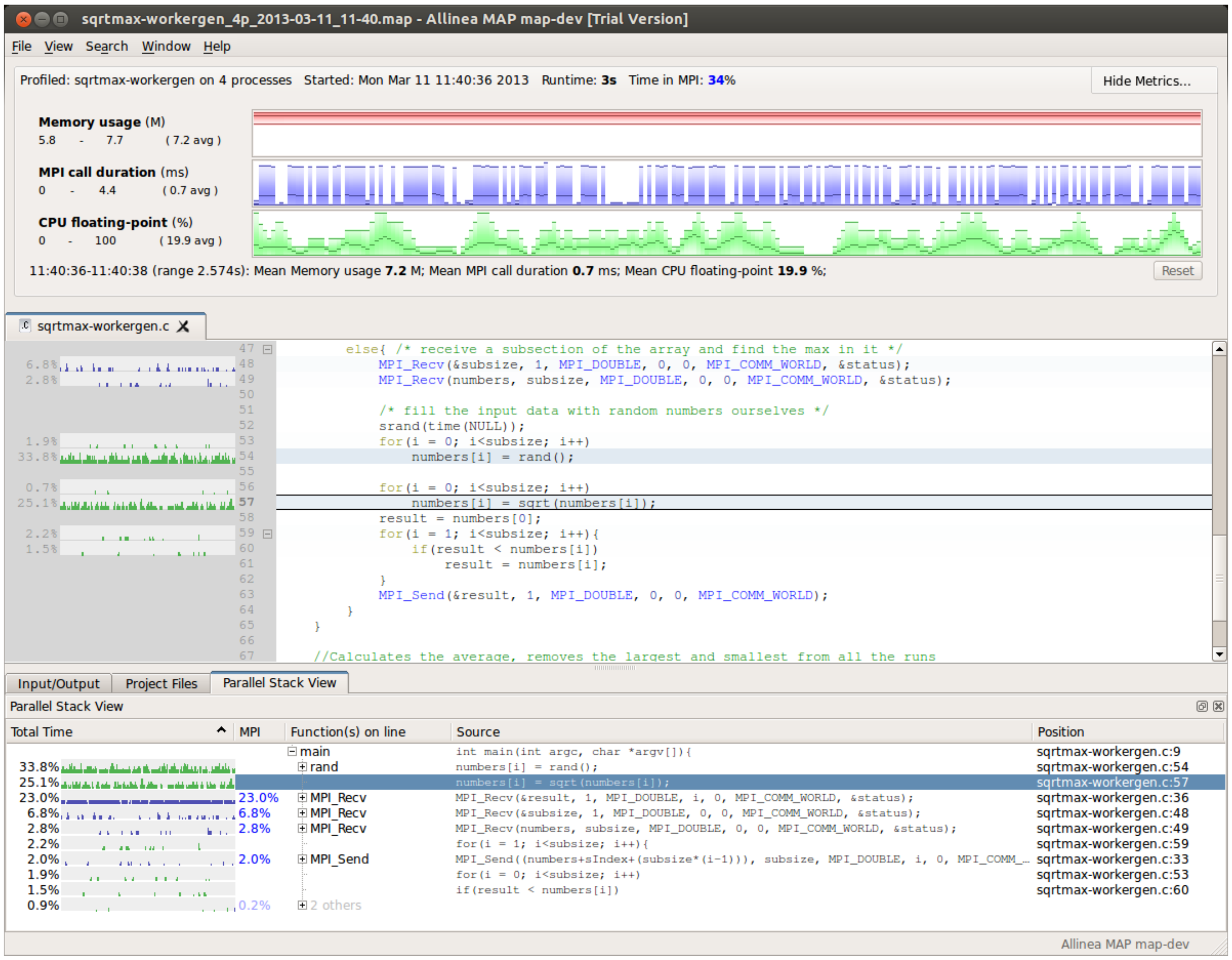
You can do this many times to gather multiple runs of different sizes or with different parameters. You can also use the -output flag to send output to a specific file, e.g:

```
      map -profile -output big-slow-run.map -n 1024 ./my_slow_program.exe
```

To view a saved file, simply pass it as an argument to map (tab-complete is your friend):

```
~/2-cpu-optimization/problem $ map sqrtmax-workergen_4p_2013-03-11_11-40.map
```

This is what MAP shows:



sqrtmax-workergen_4p_2013-03-11_11-40.map - Allinea MAP map-dev [Trial Version]

Profiled: sqrtmax-workergen on 4 processes Started: Mon Mar 11 11:40:36 2013 Runtime: **3s** Time in MPI: **34**%

Memory usage (M) 5.8 - 7.7 (7.2 avg)

MPI call duration (ms) 0 - 4.4 (0.7 avg)

CPU floating-point (%) 0 - 100 (19.9 avg)

11:40:36-11:40:38 (range 2.574s): Mean Memory usage **7.2** M; Mean MPI call duration **0.7** ms; Mean CPU floating-point **19.9** %;

sqrtmax-workergen.c

```
        else{ /* receive a subsection of the array and find the max in it */
            MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
            MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

            /* fill the input data with random numbers ourselves */
            srand(time(NULL));
            for(i = 0; i<subsize; i++)
                numbers[i] = rand();

            for(i = 0; i<subsize; i++)
                numbers[i] = sqrt(numbers[i]);
            result = numbers[0];
            for(i = 1; i<subsize; i++){
                if(result < numbers[i])
                    result = numbers[i];
            }
            MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }
    }

    //Calculates the average, removes the largest and smallest from all the runs
```

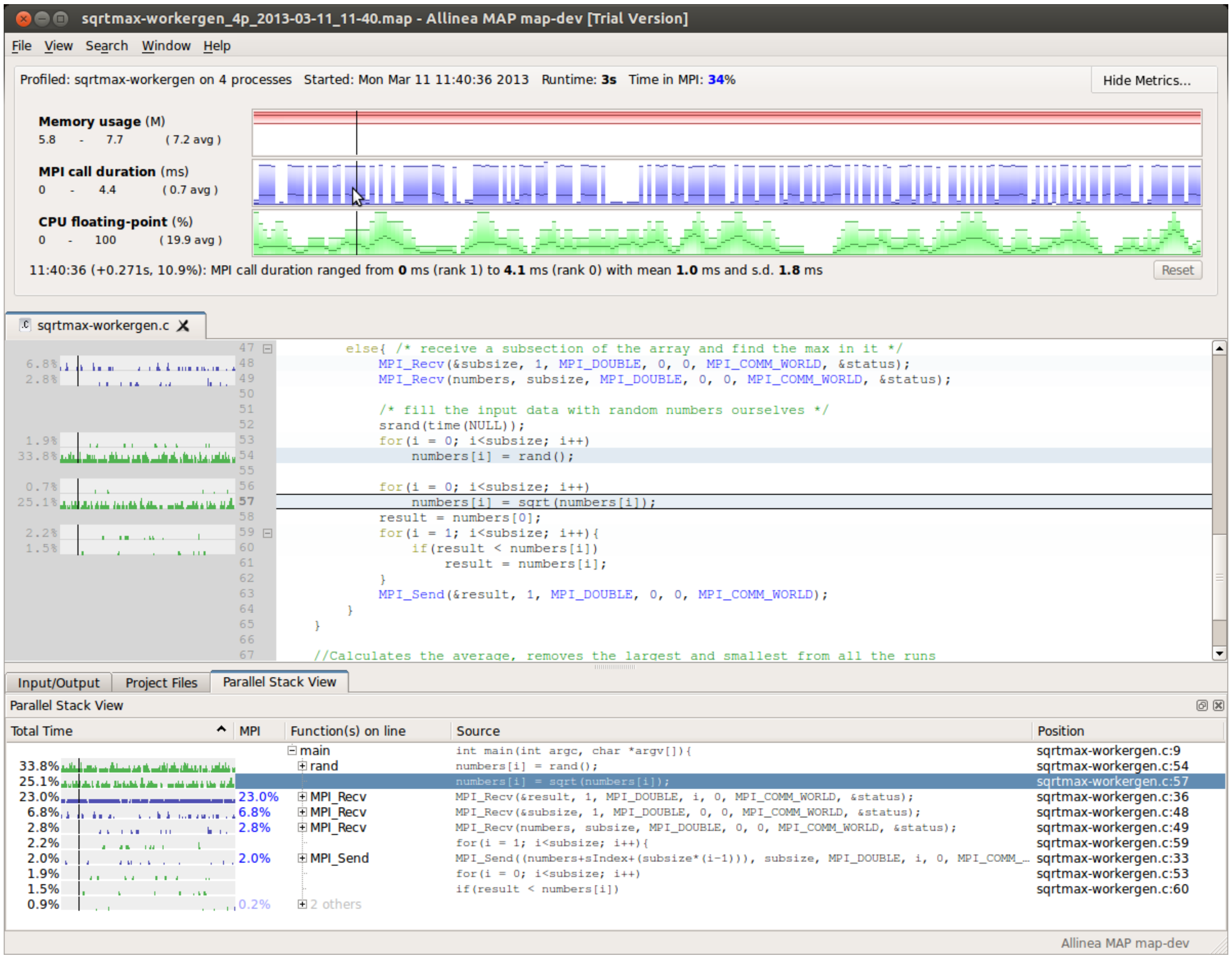
Parallel Stack View

Total Time	MPI	Function(s) on line	Source	Position
		main	`int main(int argc, char *argv[]){`	sqrtmax-workergen.c:9
33.8%		rand	`numbers[i] = rand();`	sqrtmax-workergen.c:54
25.1%			`numbers[i] = sqrt(numbers[i]);`	sqrtmax-workergen.c:57
23.0%	23.0%	MPI_Recv	`MPI_Recv(&result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:36
6.8%	6.8%	MPI_Recv	`MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:48
2.8%	2.8%	MPI_Recv	`MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:49
2.2%			`for(i = 1; i<subsize; i++){`	sqrtmax-workergen.c:59
2.0%	2.0%	MPI_Send	`MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_...`	sqrtmax-workergen.c:33
1.9%			`for(i = 0; i<subsize; i++)`	sqrtmax-workergen.c:53
1.5%			`if(result < numbers[i])`	sqrtmax-workergen.c:60
0.9%	0.2%	2 others		

Allinea MAP map-dev

We can see that the time spent in MPI is only 34% (top row). Let's look at the three big metric graphs at the top. Like the small sparkline-style charts shown next to the source code viewer, the x-axis is time. The y-axis is the metric of interest, e.g. % time spent executing floating-point instructions. Unlinke the sparkline charts, these graphs show the distribution of values across all processes. In each graph the top-most line is the maximum value across all processes, the bottom-most is the minimum value and the darker middle line is the mean value. The shading between the max and min lines represents the variance of the distribution.

In the MPI call duration graph we can see that the min and mean are close together, but the max extends a long way up. This indicates the presence of outliers, of unusually-high variance between processes. Hovering the mouse over this graph shows the min and max ranks in the text beneath:


sqrtmax-workergen_4p_2013-03-11_11-40.map - Allinea MAP map-dev [Trial Version]
File View Search Window Help
Hide Metrics...
Memory usage (M) 5.8 - 7.7 (7.2 avg)
MPI call duration (ms) 0 - 4.4 (0.7 avg)
CPU floating-point (%) 0 - 100 (19.9 avg)
Reset
sqrtmax-workergen.c
Input/Output
Project Files
Parallel Stack View
Allinea MAP map-dev

Profiled: sqrtmax-workergen on 4 processes Started: Mon Mar 11 11:40:36 2013 Runtime: **3s** Time in MPI: **34**%

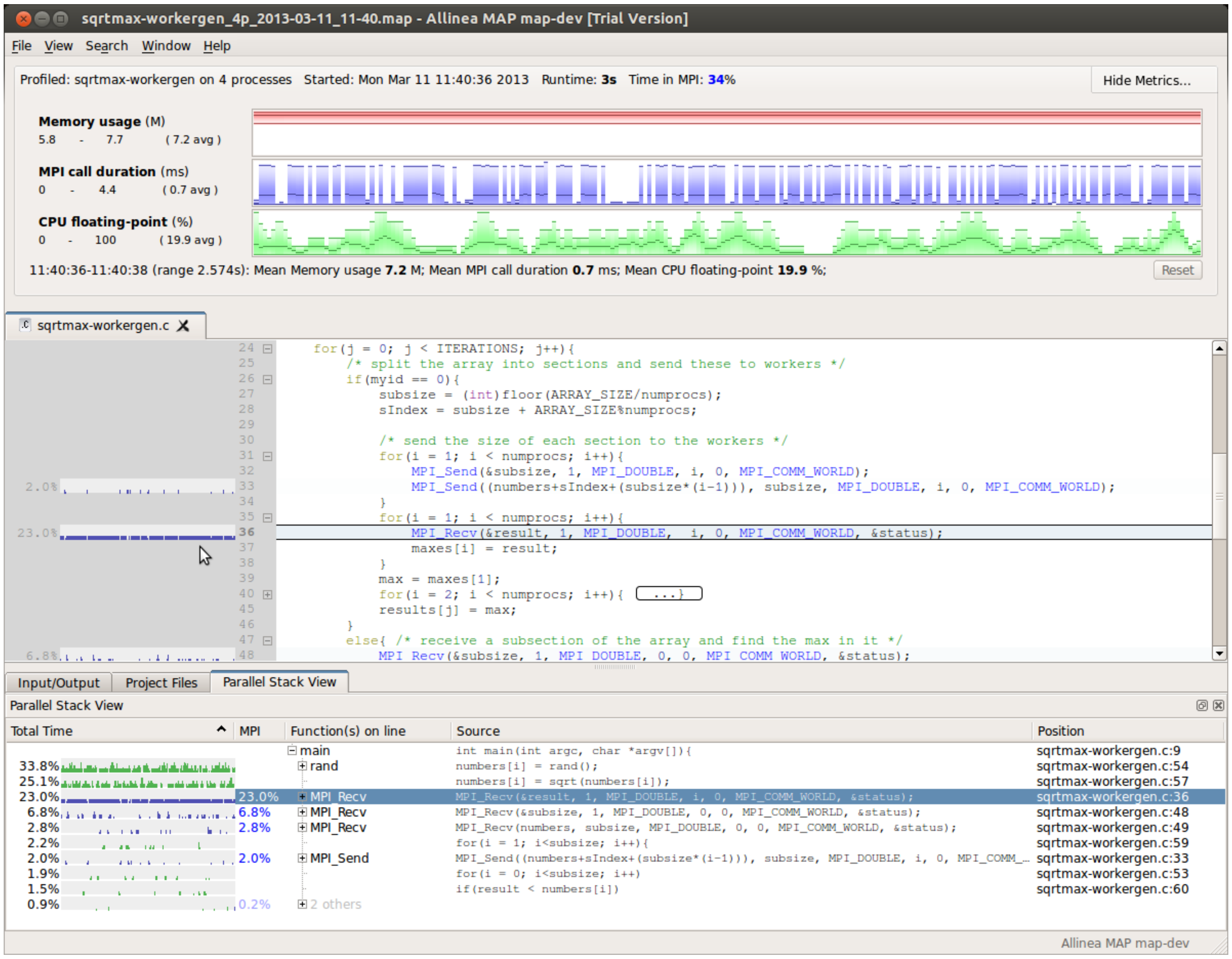
11:40:36 (+0.271s, 10.9%): MPI call duration ranged from **0** ms (rank 1) to **4.1** ms (rank 0) with mean **1.0** ms and s.d. **1.8** ms

```
        else{ /* receive a subsection of the array and find the max in it */
            MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
            MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

            /* fill the input data with random numbers ourselves */
            srand(time(NULL));
            for(i = 0; i<subsize; i++)
                numbers[i] = rand();

            for(i = 0; i<subsize; i++)
                numbers[i] = sqrt(numbers[i]);
            result = numbers[0];
            for(i = 1; i<subsize; i++){
                if(result < numbers[i])
                    result = numbers[i];
            }
            MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }
    }

    //Calculates the average, removes the largest and smallest from all the runs
```

Parallel Stack View

Total Time	MPI	Function(s) on line	Source	Position
		⊟ main	`int main(int argc, char *argv[]){`	sqrtmax-workergen.c:9
33.8%		⊞ rand	`numbers[i] = rand();`	sqrtmax-workergen.c:54
25.1%			`numbers[i] = sqrt(numbers[i]);`	sqrtmax-workergen.c:57
23.0%	23.0%	⊞ MPI_Recv	`MPI_Recv(&result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:36
6.8%	6.8%	⊞ MPI_Recv	`MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:48
2.8%	2.8%	⊞ MPI_Recv	`MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:49
2.2%			`for(i = 1; i<subsize; i++){`	sqrtmax-workergen.c:59
2.0%	2.0%	⊞ MPI_Send	`MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_…`	sqrtmax-workergen.c:33
1.9%			`for(i = 0; i<subsize; i++)`	sqrtmax-workergen.c:53
1.5%			`if(result < numbers[i])`	sqrtmax-workergen.c:60
0.9%	0.2%	⊞ 2 others		

In this case, at pretty much any point along the graph rank 0 spends the most time in MPI calls. This is reflected by the large amount of time spent on the MPI_Recv line inside the if(myid==0) branch:



sqrtmax-workergen_4p_2013-03-11_11-40.map - Allinea MAP map-dev [Trial Version]

Profiled: sqrtmax-workergen on 4 processes Started: Mon Mar 11 11:40:36 2013 Runtime: **3s** Time in MPI: **34%**

Memory usage (M) 5.8 - 7.7 (7.2 avg)

MPI call duration (ms) 0 - 4.4 (0.7 avg)

CPU floating-point (%) 0 - 100 (19.9 avg)

11:40:36-11:40:38 (range 2.574s): Mean Memory usage **7.2** M; Mean MPI call duration **0.7** ms; Mean CPU floating-point **19.9** %;

sqrtmax-workergen.c

```
    for(j = 0; j < ITERATIONS; j++){
        /* split the array into sections and send these to workers */
        if(myid == 0){
            subsize = (int)floor(ARRAY_SIZE/numprocs);
            sIndex = subsize + ARRAY_SIZE%numprocs;

            /* send the size of each section to the workers */
            for(i = 1; i < numprocs; i++){
                MPI_Send(&subsize, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
                MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
            }
            for(i = 1; i < numprocs; i++){
                MPI_Recv(&result, 1, MPI_DOUBLE,  i, 0, MPI_COMM_WORLD, &status);
                maxes[i] = result;
            }
            max = maxes[1];
            for(i = 2; i < numprocs; i++){ ...}
            results[j] = max;
        }
        else{ /* receive a subsection of the array and find the max in it */
            MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
```

Parallel Stack View

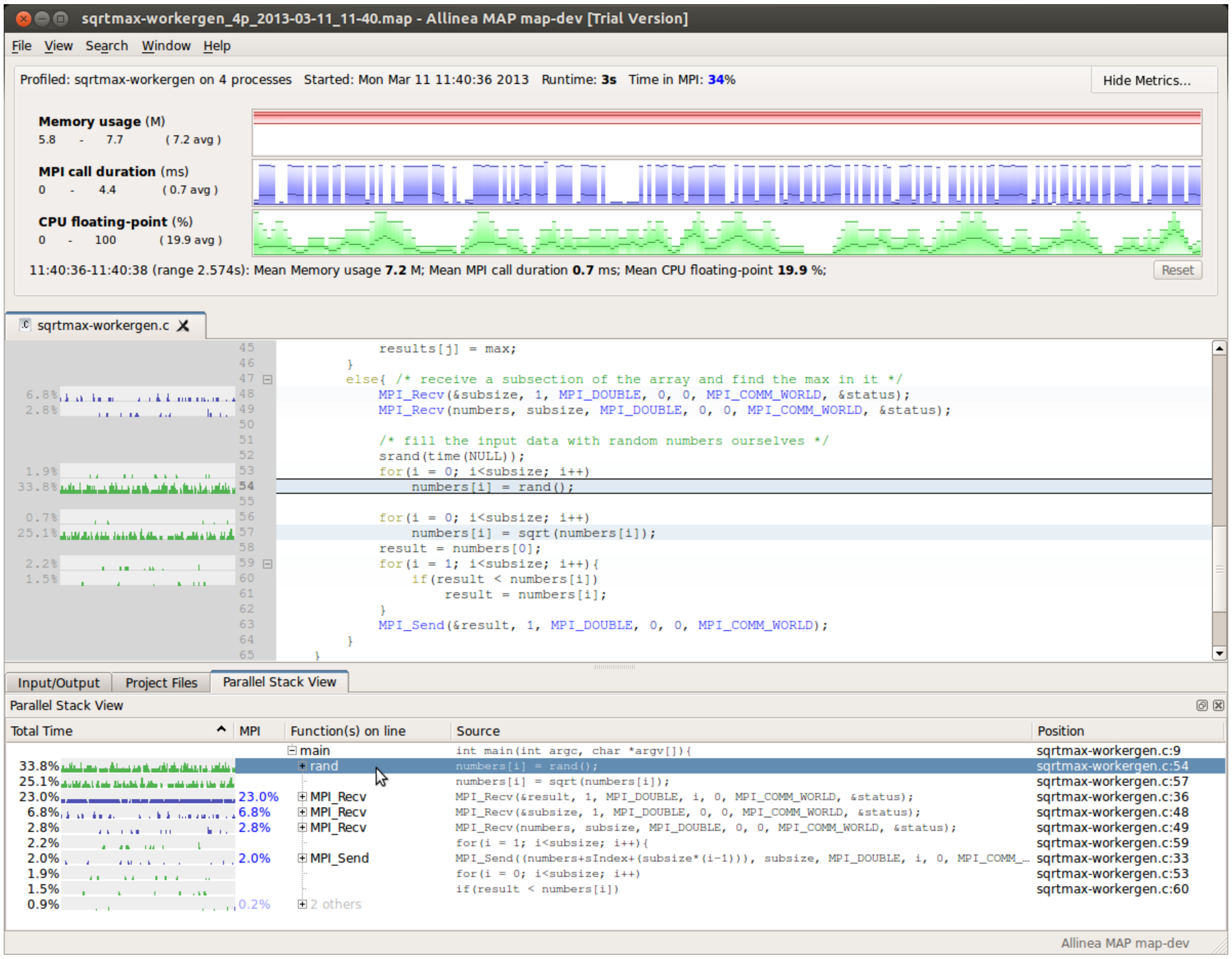
Total Time	MPI	Function(s) on line	Source	Position
		main	`int main(int argc, char *argv[]){`	sqrtmax-workergen.c:9
33.8%		rand	`numbers[i] = rand();`	sqrtmax-workergen.c:54
25.1%			`numbers[i] = sqrt(numbers[i]);`	sqrtmax-workergen.c:57
23.0%	23.0%	MPI_Recv	`MPI_Recv(&result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:36
6.8%	6.8%	MPI_Recv	`MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:48
2.8%	2.8%	MPI_Recv	`MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:49
2.2%			`for(i = 1; i<subsize; i++){`	sqrtmax-workergen.c:59
2.0%	2.0%	MPI_Send	`MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_…`	sqrtmax-workergen.c:33
1.9%			`for(i = 0; i<subsize; i++)`	sqrtmax-workergen.c:53
1.5%			`if(result < numbers[i])`	sqrtmax-workergen.c:60
0.9%	0.2%	2 others		

Allinea MAP map-dev

We can clearly see that rank 0 doesn't take part in the computation but rather spends all its time waiting for the results from the other ranks.

While we could get around a 25% speedup here at 4 processes, this will become less significant as the size of the job increases, so we're not going to spend time on it right now. Instead, look at the CPU floating-point % graph. Our cores are only spending 20% of their time doing floating-point calculations, which are typically what they were bought for! In this case, our workload is a combination of sqrt and max, so we'd expect to see higher usage here. Why don't we?

The Parallel Stack View at the bottom breaks down the time spent by function. We can see that both rand() and sqrt() feature prominently. In fact, we spend more time in rand() than we do in sqrt() - 33% vs 25%! Clicking on rand() takes the source code viewer to the site of the call:


sqrtmax-workergen_4p_2013-03-11_11-40.map - Allinea MAP map-dev [Trial Version]
File View Search Window Help
Hide Metrics...
Reset
sqrtmax-workergen.c
Input/Output Project Files Parallel Stack View
Allinea MAP map-dev

Profiled: sqrtmax-workergen on 4 processes Started: Mon Mar 11 11:40:36 2013 Runtime: **3s** Time in MPI: **34**%

Memory usage (M) 5.8 - 7.7 (7.2 avg)

MPI call duration (ms) 0 - 4.4 (0.7 avg)

CPU floating-point (%) 0 - 100 (19.9 avg)

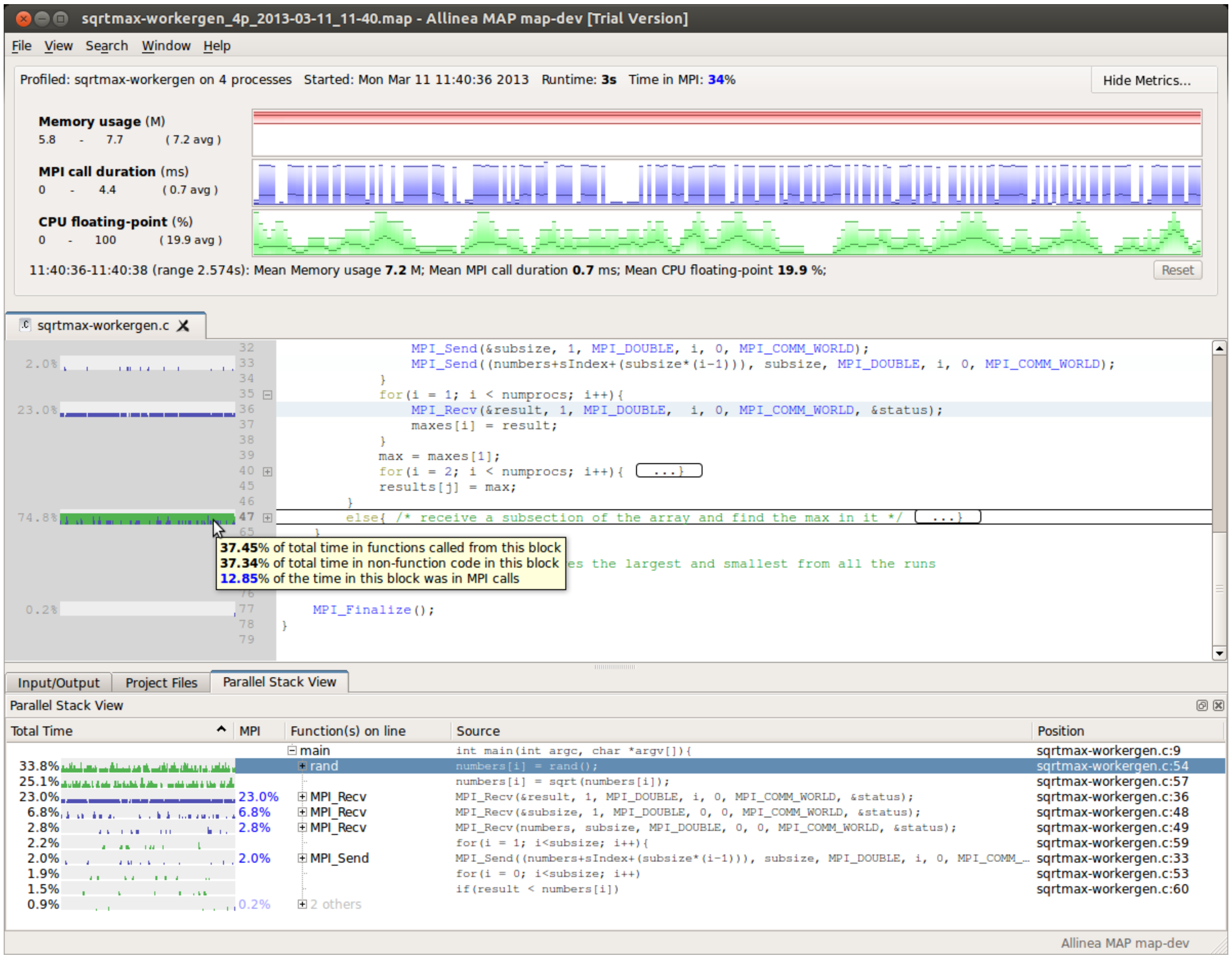
11:40:36-11:40:38 (range 2.574s): Mean Memory usage **7.2** M; Mean MPI call duration **0.7** ms; Mean CPU floating-point **19.9** %;

```
        results[j] = max;
    }
    else{ /* receive a subsection of the array and find the max in it */
        MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);      6.8%
        MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status); 2.8%

        /* fill the input data with random numbers ourselves */
        srand(time(NULL));
        for(i = 0; i<subsize; i++)                                             1.9%
            numbers[i] = rand();                                              33.8%

        for(i = 0; i<subsize; i++)                                             0.7%
            numbers[i] = sqrt(numbers[i]);                                    25.1%
        result = numbers[0];
        for(i = 1; i<subsize; i++){                                            2.2%
            if(result < numbers[i])                                            1.5%
                result = numbers[i];
        }
        MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
}
```

Parallel Stack View

Total Time	MPI	Function(s) on line	Source	Position
		⊟ main	int main(int argc, char *argv[]){	sqrtmax-workergen.c:9
33.8%		⊞ rand	numbers[i] = rand();	sqrtmax-workergen.c:54
25.1%			numbers[i] = sqrt(numbers[i]);	sqrtmax-workergen.c:57
23.0%	23.0%	⊞ MPI_Recv	MPI_Recv(&result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);	sqrtmax-workergen.c:36
6.8%	6.8%	⊞ MPI_Recv	MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);	sqrtmax-workergen.c:48
2.8%	2.8%	⊞ MPI_Recv	MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);	sqrtmax-workergen.c:49
2.2%			for(i = 1; i<subsize; i++){	sqrtmax-workergen.c:59
2.0%	2.0%	⊞ MPI_Send	MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_...	sqrtmax-workergen.c:33
1.9%			for(i = 0; i<subsize; i++)	sqrtmax-workergen.c:53
1.5%			if(result < numbers[i])	sqrtmax-workergen.c:60
0.9%	0.2%	⊞ 2 others		

Here we see each worker core generating random numbers in a loop and then taking their square root in the next loop. We can collapse this entire worker branch of the if statement down to see its overall time by clicking on the small [-] next to the 'else' statement:


sqrtmax-workergen_4p_2013-03-11_11-40.map - Allinea MAP map-dev [Trial Version]
File View Search Window Help
Hide Metrics...
Reset
sqrtmax-workergen.c
Input/Output
Project Files
Parallel Stack View
Allinea MAP map-dev

Profiled: sqrtmax-workergen on 4 processes Started: Mon Mar 11 11:40:36 2013 Runtime: 3s Time in MPI: 34%

Memory usage (M) 5.8 - 7.7 (7.2 avg)

MPI call duration (ms) 0 - 4.4 (0.7 avg)

CPU floating-point (%) 0 - 100 (19.9 avg)

11:40:36-11:40:38 (range 2.574s): Mean Memory usage **7.2** M; Mean MPI call duration **0.7** ms; Mean CPU floating-point **19.9** %;

```
                MPI_Send(&subsize, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
2.0%            MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
            }
            for(i = 1; i < numprocs; i++){
23.0%           MPI_Recv(&result, 1, MPI_DOUBLE,  i, 0, MPI_COMM_WORLD, &status);
                maxes[i] = result;
            }
            max = maxes[1];
            for(i = 2; i < numprocs; i++){ ...}
            results[j] = max;
        }
74.8%   else{ /* receive a subsection of the array and find the max in it */ ...}
    }
                                                 es the largest and smallest from all the runs

0.2%    MPI_Finalize();
}
```

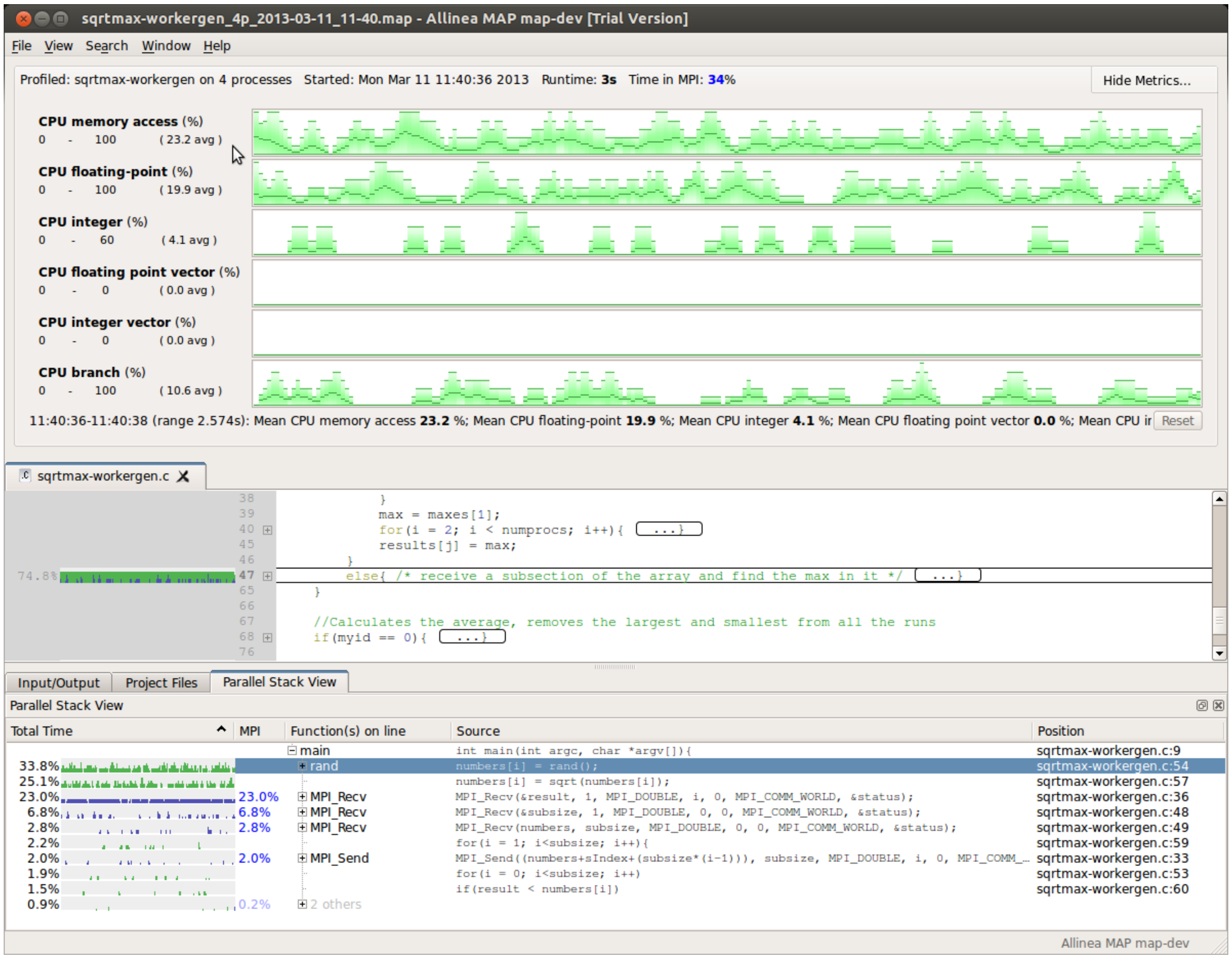
37.45% of total time in functions called from this block
37.34% of total time in non-function code in this block
12.85% of the time in this block was in MPI calls

Parallel Stack View

Total Time	MPI	Function(s) on line	Source	Position
		main	int main(int argc, char *argv[]){	sqrtmax-workergen.c:9
33.8%		rand	numbers[i] = rand();	sqrtmax-workergen.c:54
25.1%			numbers[i] = sqrt(numbers[i]);	sqrtmax-workergen.c:57
23.0%	23.0%	MPI_Recv	MPI_Recv(&result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);	sqrtmax-workergen.c:36
6.8%	6.8%	MPI_Recv	MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);	sqrtmax-workergen.c:48
2.8%	2.8%	MPI_Recv	MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);	sqrtmax-workergen.c:49
2.2%			for(i = 1; i<subsize; i++){	sqrtmax-workergen.c:59
2.0%	2.0%	MPI_Send	MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_...	sqrtmax-workergen.c:33
1.9%			for(i = 0; i<subsize; i++)	sqrtmax-workergen.c:53
1.5%			if(result < numbers[i])	sqrtmax-workergen.c:60
0.9%	0.2%	2 others		

We can now see that 75% of the time is spent inside this block of code, of which only 12.85% is in MPI functions. Clearly if we want to optimize this, we need to use the CPU more effectively.

To dig down into the CPU usage, right-click anywhere on the three big metric graphs at the top and choose "Preset: CPU Instructions" from the menu. Now we can see how much time is spent in different classes of CPU instruction:


sqrtmax-workergen_4p_2013-03-11_11-40.map - Allinea MAP map-dev [Trial Version]
File View Search Window Help
Hide Metrics...
CPU memory access (%) 0 - 100 (23.2 avg)
CPU floating-point (%) 0 - 100 (19.9 avg)
CPU integer (%) 0 - 60 (4.1 avg)
CPU floating point vector (%) 0 - 0 (0.0 avg)
CPU integer vector (%) 0 - 0 (0.0 avg)
CPU branch (%) 0 - 100 (10.6 avg)
Reset
sqrtmax-workergen.c
74.8%
Input/Output
Project Files
Parallel Stack View
Allinea MAP map-dev

Profiled: sqrtmax-workergen on 4 processes Started: Mon Mar 11 11:40:36 2013 Runtime: **3s** Time in MPI: **34**%

11:40:36-11:40:38 (range 2.574s): Mean CPU memory access **23.2** %; Mean CPU floating-point **19.9** %; Mean CPU integer **4.1** %; Mean CPU floating point vector **0.0** %; Mean CPU ir

```
            }
            max = maxes[1];
            for(i = 2; i < numprocs; i++){ ...}
            results[j] = max;
        }
        else{ /* receive a subsection of the array and find the max in it */ ...}
    }

    //Calculates the average, removes the largest and smallest from all the runs
    if(myid == 0){ ...}
```

Parallel Stack View

Total Time	MPI	Function(s) on line	Source	Position
		main	`int main(int argc, char *argv[]){`	sqrtmax-workergen.c:9
33.8%		rand	`numbers[i] = rand();`	sqrtmax-workergen.c:54
25.1%			`numbers[i] = sqrt(numbers[i]);`	sqrtmax-workergen.c:57
23.0%	23.0%	MPI_Recv	`MPI_Recv(&result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:36
6.8%	6.8%	MPI_Recv	`MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:48
2.8%	2.8%	MPI_Recv	`MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:49
2.2%			`for(i = 1; i<subsize; i++){`	sqrtmax-workergen.c:59
2.0%	2.0%	MPI_Send	`MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, MPI_COMM_…`	sqrtmax-workergen.c:33
1.9%			`for(i = 0; i<subsize; i++)`	sqrtmax-workergen.c:53
1.5%			`if(result < numbers[i])`	sqrtmax-workergen.c:60
0.9%	0.2%	2 others		

The most important thing to notice here is that no vectorized CPU instructions are being used at all. These are the packed SSE2/AVX instructions generated by most vectorizing compilers. They are absolutely critical to even approaching the peak throughput of modern CPUs, and the lack of any is a clear red flag. In this case, a vectorized version of sqrt() isn't being used. A quick online search tells us we need to pass the -ffast-math flag to GCC to enable this:

```
mark@Namaste-II ~/2-cpu-optimization/problem $ mpicc -g -O3 -ffast-math sqrtmax-
workergen.c -o sqrtmax-vector

mark@Namaste-II ~/2-cpu-optimization/problem $ time mpirun -n 4 ./sqrtmax-vector
Max after 500 iterations:46340.9

real    0m3.058s
user    0m8.133s
sys     0m0.020s
```

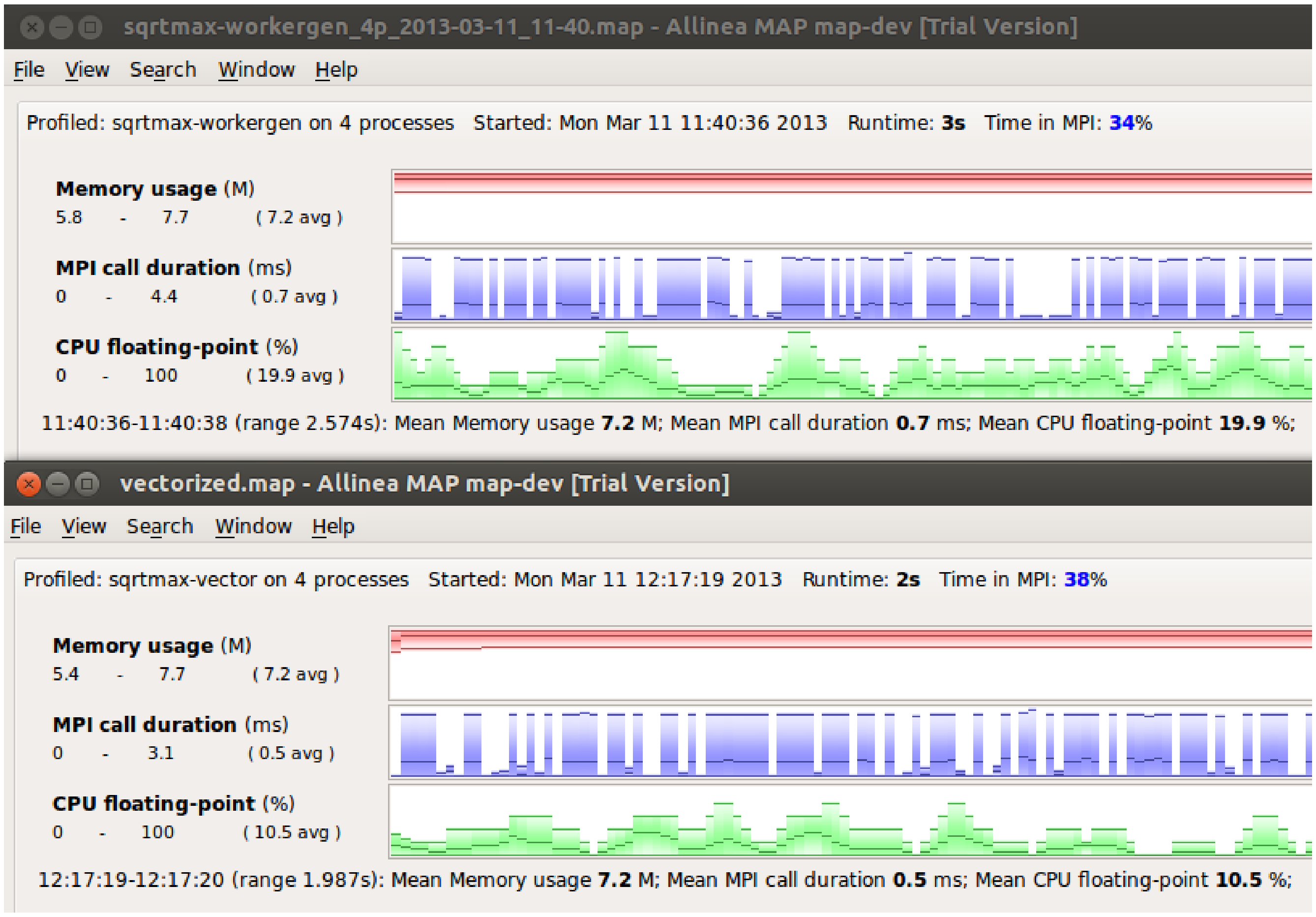
This alone grants us an extra 15% speedup from the previous runtime of 3.620s and that's including the mpirun overhead which will be seriously distorting the figures at such low runtimes. Let's run the new version in MAP:

```
mark@Namaste-II ~/2-cpu-optimization/problem $ map -profile -output vectorized.map
-n 4 ./sqrtmax-vector
```

```
Allinea MAP map-dev, (c) Allinea Software Ltd
Profiling                    : /home/mark/Work/code/doc/training/map-
introduction/2-cpu-optimization/problem/sqrtmax-vector
Processes                    : 4
MPI                          : OpenMPI
Statically linked MPI wrapper: no
Static linked sampler        : no

MAP starting profiling...
mpirun: Max after 500 iterations:46340.9

MAP analysing program...
MAP gathering samples...
MAP generated vectorized.map
```

In MAP, the runtime is measured from the end of MPI_Init to the moment the last process reaches MPI_Finalize. This cuts out the overhead of starting the MPI job and shutting it down again, providing a better estimate for the speed of the code and its performance on larger data sets. We'll just open both files for a quick look:

```
mark@Namaste-II ~/2-cpu-optimization/problem $ map sqrtmax-workergen_4p_2013-03-
11_11-40.map & map vectorized.map
```


sqrtmax-workergen_4p_2013-03-11_11-40.map - Allinea MAP map-dev [Trial Version]
File View Search Window Help
vectorized.map - Allinea MAP map-dev [Trial Version]
File View Search Window Help

Profiled: sqrtmax-workergen on 4 processes   Started: Mon Mar 11 11:40:36 2013   Runtime: **3s**   Time in MPI: **34**%

Memory usage (M)
5.8 - 7.7 (7.2 avg)

MPI call duration (ms)
0 - 4.4 (0.7 avg)

CPU floating-point (%)
0 - 100 (19.9 avg)

11:40:36-11:40:38 (range 2.574s): Mean Memory usage **7.2** M; Mean MPI call duration **0.7** ms; Mean CPU floating-point **19.9** %;

Profiled: sqrtmax-vector on 4 processes   Started: Mon Mar 11 12:17:19 2013   Runtime: **2s**   Time in MPI: **38**%

Memory usage (M)
5.4 - 7.7 (7.2 avg)

MPI call duration (ms)
0 - 3.1 (0.5 avg)

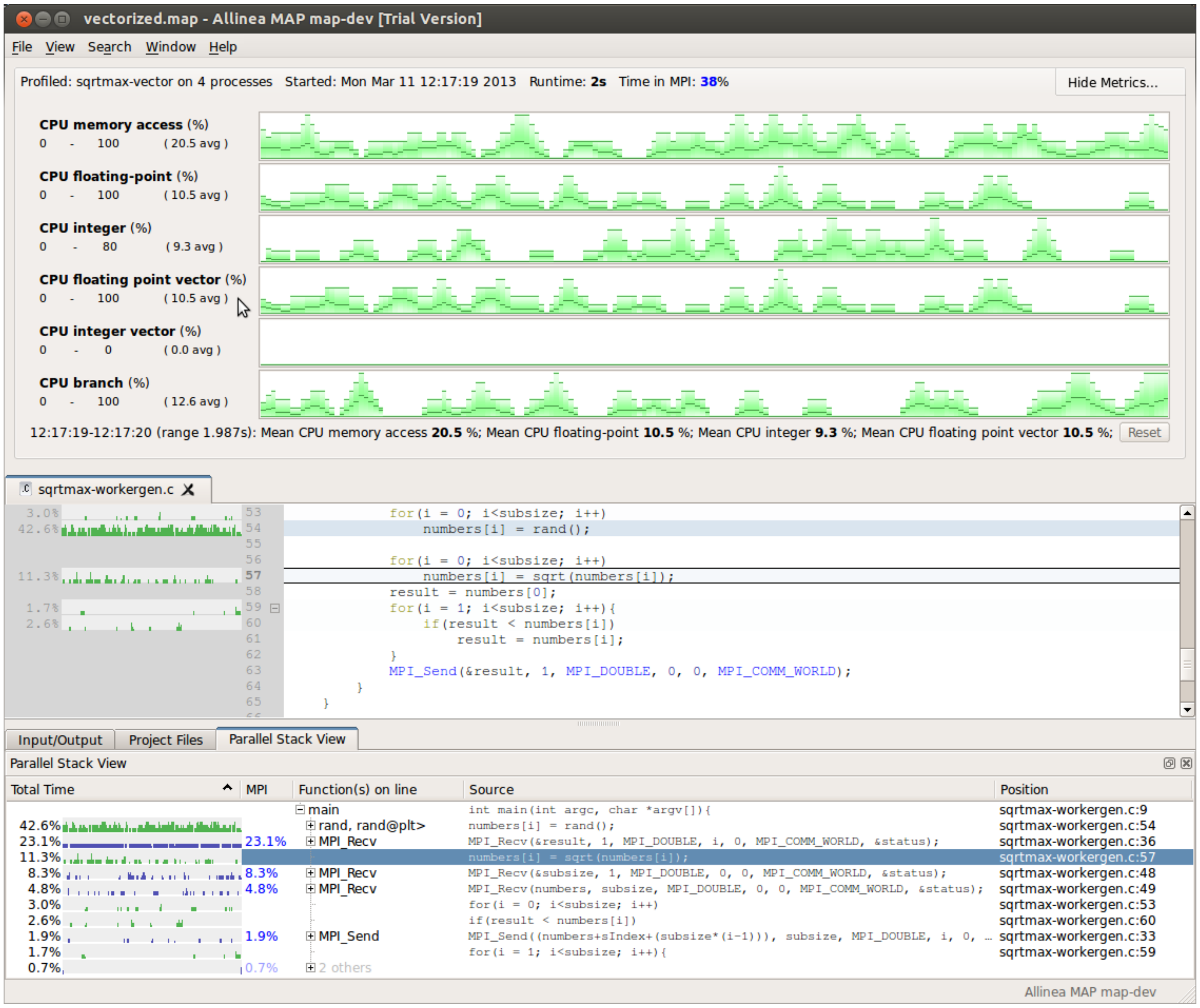
CPU floating-point (%)
0 - 100 (10.5 avg)

12:17:19-12:17:20 (range 1.987s): Mean Memory usage **7.2** M; Mean MPI call duration **0.5** ms; Mean CPU floating-point **10.5** %;

In this case it tells us that the non-vectorized version took 2.574s and the vectorized one just 1.987s.

That's a speedup of 23% just by enabling vectorization! Notice that the % time spent in CPU floating-point instructions has actually decreased from 19.9% to just 10.5% - this is because the vectorized instructions are much more efficient, so the CPU spends less time overall executing them.

Close the non-vectorized version and focus on vectorized.map. Right-click on one of the large metrics and choose "Preset: CPU Instructions" again:



Profiled: sqrtmax-vector on 4 processes Started: Mon Mar 11 12:17:19 2013 Runtime: **2s** Time in MPI: 38%

Metric	Range	Average
CPU memory access (%)	0 - 100	(20.5 avg)
CPU floating-point (%)	0 - 100	(10.5 avg)
CPU integer (%)	0 - 80	(9.3 avg)
CPU floating point vector (%)	0 - 100	(10.5 avg)
CPU integer vector (%)	0 - 0	(0.0 avg)
CPU branch (%)	0 - 100	(12.6 avg)

12:17:19-12:17:20 (range 1.987s): Mean CPU memory access **20.5** %; Mean CPU floating-point **10.5** %; Mean CPU integer **9.3** %; Mean CPU floating point vector **10.5** %;

```
3.0%         for(i = 0; i<subsize; i++)
42.6%            numbers[i] = rand();

             for(i = 0; i<subsize; i++)
11.3%            numbers[i] = sqrt(numbers[i]);
             result = numbers[0];
1.7%         for(i = 1; i<subsize; i++){
2.6%             if(result < numbers[i])
                     result = numbers[i];
             }
             MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
         }
     }
```

Parallel Stack View

Total Time	MPI	Function(s) on line	Source	Position
		⊟ main	`int main(int argc, char *argv[]){`	sqrtmax-workergen.c:9
42.6%		⊞ rand, rand@plt>	`numbers[i] = rand();`	sqrtmax-workergen.c:54
23.1%	23.1%	⊞ MPI_Recv	`MPI_Recv(&result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:36
11.3%			`numbers[i] = sqrt(numbers[i]);`	sqrtmax-workergen.c:57
8.3%	8.3%	⊞ MPI_Recv	`MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:48
4.8%	4.8%	⊞ MPI_Recv	`MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);`	sqrtmax-workergen.c:49
3.0%			`for(i = 0; i<subsize; i++)`	sqrtmax-workergen.c:53
2.6%			`if(result < numbers[i])`	sqrtmax-workergen.c:60
1.9%	1.9%	⊞ MPI_Send	`MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize, MPI_DOUBLE, i, 0, …`	sqrtmax-workergen.c:33
1.7%			`for(i = 1; i<subsize; i++){`	sqrtmax-workergen.c:59
0.7%	0.7%	⊞ 2 others		

Now we can see that the percentage of time spent executing vectorized floating-point instructions is roughly the same as on all floating-point instructions. In fact, the two charts are exact duplicates of one another. This is a good sign - our floating-point workload is now well-vectorized.

The most significant consumer of CPU time here is memory accesses, seen at the top. Memory accesses take up around 20% of the CPU's time, double that of the floating-point instructions. This is a clear sign of poor cache performance - whenever your CPU spends more time fetching numbers than crunching them, there's usually room for some serious improvement.

Cache optimization techniques could have a one week training course all to themselves, but the basics boil down to three things:

1. If you see a pattern like this in your code (higher % time in memory instructions than in floating-point instructions) and have access to a single-core performance expert, show them the graphs and affected loops and ask them to take a look at it. An expert can often get 5x or even 20x the performance out of a loop, but it's quite a specialized art.

While you're waiting for them to reply to your email, there's nothing wrong with having a quick look at it yourself. Cache performance is about locality:

2. Spatial locality. When the CPU accesses values from an array, it copies an entire chunk of the array into cache with it in the expectation that the nearby values will also be used soon afterwards. To make good use of the cache, therefore, we want to use those values. The classic example of this going wrong is with a nested loop over a two-dimensional array. If you iterate in one direction (rows then columns) or the other (columns then rows) then it's easy to fill the cache with values that aren't used. An example of this can be seen in MAP's examples/slow.f90 - look at the stride() function. In short, if a nested loop is showing poor cache performance, try swapping the order of the nesting and see if it improves.

3. Temporal locality. CPU cache is a limited resource and the values will only be in there for a short window. It's best to use them as often as possible iwithin that window before moving on to the next values. In practice, this means fusing loops. To illustrate this, let's look at the example code again. Here we have two loops - the first generates random numbers and saves them into the numbers array. The second loads a number from the array, calculates its square root and saves it back into the array again (lines 53-57 in the previous figure).

We can improve this by joining these two loops together, so that we iterate once through the array, generate the random number and its square root at the same time.

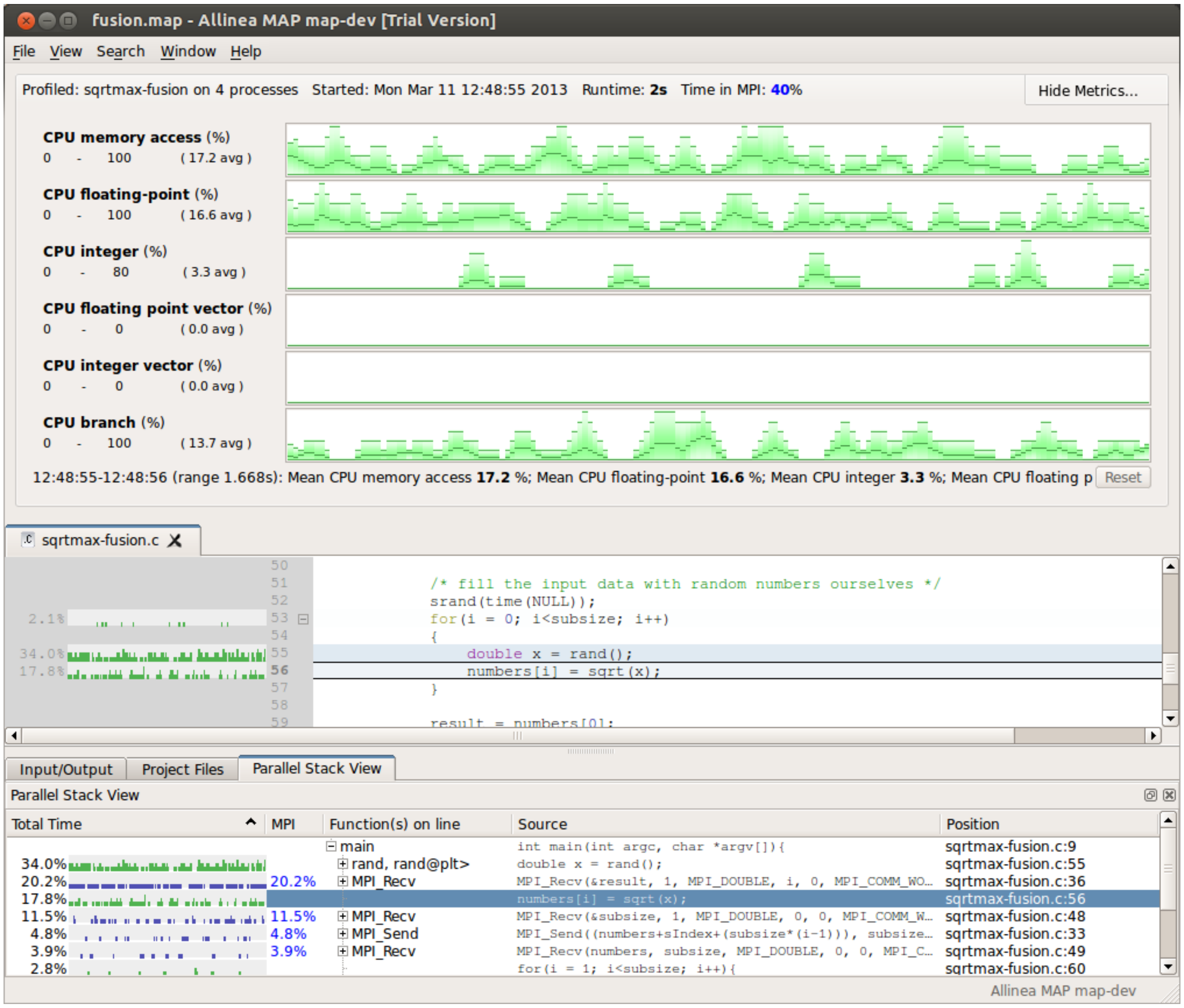
Exercise 1: Fuse these two loops together into one, then recompile the example (remember to use -ffast-math for vectorization) and measure its speed. If the speed up isn't as much as you were expecting, remember to use MAP to compare the runtimes instead of 'time mpirun'! How much faster is the version with fused loops? How much have we improved the performance in total this session?

Answers: One way to fuse the loops is shown in solution/sqrtmax-fusion.c - simply replace both loops with one that does "double x = rand(); numbers[i] = sqrt(x);". This is preferred to calling sqrt(rand()) because:

1. *The compiler will generate the same code for both versions, temporary variable elimination is trivial these days.*
2. *Splitting the operations out allows us to see in MAP how much time was spent on each line, which in this case is enlightening:*

   After the loop fusion, the program takes just 1.668s on my laptop between MPI_Init and MPI_Finalize (see figure below). The % floating-point instructions is up to 16.6%, similar to the % time in memory accesses (17.2%). This represents a 16% improvement in runtime compared to the vectorized example and a full 35% improvement for this session alone.

Although the fusion example is faster, look at the vectorization results in MAP:



Profiled: sqrtmax-fusion on 4 processes Started: Mon Mar 11 12:48:55 2013 Runtime: **2s** Time in MPI: **40%**

Metric	Range	Average
CPU memory access (%)	0 - 100	(17.2 avg)
CPU floating-point (%)	0 - 100	(16.6 avg)
CPU integer (%)	0 - 80	(3.3 avg)
CPU floating point vector (%)	0 - 0	(0.0 avg)
CPU integer vector (%)	0 - 0	(0.0 avg)
CPU branch (%)	0 - 100	(13.7 avg)

12:48:55-12:48:56 (range 1.668s): Mean CPU memory access **17.2** %; Mean CPU floating-point **16.6** %; Mean CPU integer **3.3** %; Mean CPU floating p

```
50
51            /* fill the input data with random numbers ourselves */
52            srand(time(NULL));
53            for(i = 0; i<subsize; i++)
54            {
55     34.0%      double x = rand();
56     17.8%      numbers[i] = sqrt(x);
57            }
58
59            result = numbers[0];
```

Parallel Stack View

Total Time	MPI	Function(s) on line	Source	Position
		main	int main(int argc, char *argv[]){	sqrtmax-fusion.c:9
34.0%		rand, rand@plt>	double x = rand();	sqrtmax-fusion.c:55
20.2%	20.2%	MPI_Recv	MPI_Recv(&result, 1, MPI_DOUBLE, i, 0, MPI_COMM_WO…	sqrtmax-fusion.c:36
17.8%			numbers[i] = sqrt(x);	sqrtmax-fusion.c:56
11.5%	11.5%	MPI_Recv	MPI_Recv(&subsize, 1, MPI_DOUBLE, 0, 0, MPI_COMM_W…	sqrtmax-fusion.c:48
4.8%	4.8%	MPI_Send	MPI_Send((numbers+sIndex+(subsize*(i-1))), subsize…	sqrtmax-fusion.c:33
3.9%	3.9%	MPI_Recv	MPI_Recv(numbers, subsize, MPI_DOUBLE, 0, 0, MPI_C…	sqrtmax-fusion.c:49
2.8%			for(i = 1; i<subsize; i++){	sqrtmax-fusion.c:60

Now we are back to 0% time in vectorized instructions! Why is this? Well, we can ask GCC directly with -ftree-vectorizer-verbose=6:

```
mark@Namaste-II ~/2-cpu-optimization/solution $ mpicc -g -O3 -ffast-math sqrtmax-
fusion.c -o sqrtmax-fusion -ftree-vectorizer-verbose=6
...
sqrtmax-fusion.c:53: note: not vectorized: number of iterations cannot be
computed.
sqrtmax-fusion.c:9: note: vectorized 2 loops in function.
```

This sort of thing is why optimization must always be followed up with measurement. Keep a .map profile for each change you make to your program so that you can go back and track unexpected side effects like this. In this case, the question begs: why can't GCC vectorize this loop now when it could before? Let's look at the output for the previous version of the program - you're all using source control, right?

```
mark@Namaste-II ~/2-cpu-optimization/problem $ mpicc -g -O3 -ffast-math sqrtmax-
workergen.c -o sqrtmax-vector -ftree-vectorizer-verbose=6
...
```

```
sqrtmax-workergen.c:56: note: Profitability threshold is 2 loop iterations.
sqrtmax-workergen.c:56: note: LOOP VECTORIZED.
sqrtmax-workergen.c:53: note: not vectorized: number of iterations cannot be
computed.
```

So here we see that GCC was able to vectorize the sqrt loop but cannot vectorize the loop containing rand(). This is probably because it is inlining rand() and cannot determine in advance how many iterations of that code will be performed.

Exercise 2: What could we do to further improve the performance of this example?

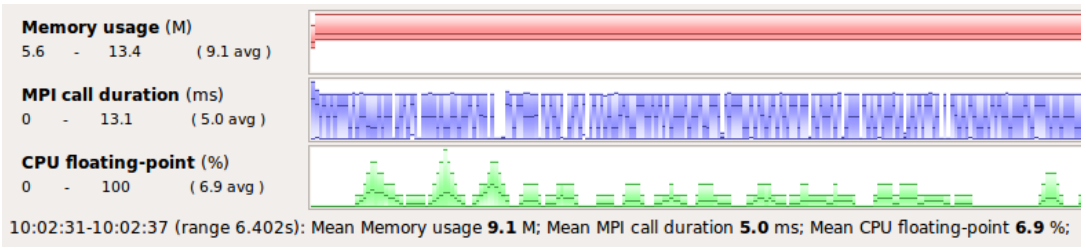
Answer: There are two good ways to go from here:

1. *Now that the program is spending 40% of its time in MPI calls it may be time to tackle this low-hanging fruit. Utilizing rank 0 for computations would bring the biggest immediate boost (just under 25%) followed by optimizing the communication pattern - the MPI_Send / MPI_Recv communications are now taking 17% of the workers' time. Good optimizations to make here include not sending the entire numbers array(!) and using MPI_Scatter to send each process its subsize and MPI_Gather or even MPI_Reduce to receive the maximums back. Optimizing the size of the numbers array (it doesn't need to be as large as it is) is not necessary or important as there's no memory pressure to be seen (just 7.7Mb of RAM).*
2. *Most of the compute time is spent calling rand(). There are other random number generators available, they should be investigated as alternatives. One example: using drand48 as a drop-in replacement improves performance slightly, greatly reduces the time spent in memory accesses but only slightly improves overall runtime. Swapping in the random() function instead drops the time to just 1.540s, a further 8% performance improvement!*

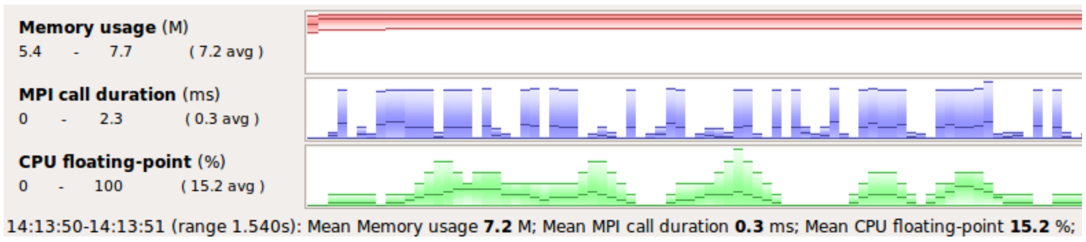
Review

In this session we've looked at single-core performance and the rich benefits to be had from optimizing it. We've covered vectorization including how to tell whether it is being used or not and have also looked at cache performance - how to detect poor cache performance and how to improve it. We also learned how to use MAP's -profile mode to collect data without the GUI and used its metric views to analyze CPU and MPI behaviour in more detail.

During the course of this we improved our example program's runtime by an impressive 35%. Combined with the previous session, we have decreased the runtime by 74% - a 4x speedup - and there are still many obvious improvements to be made:


Memory usage (M) 5.6 - 13.4 (9.1 avg)
MPI call duration (ms) 0 - 13.1 (5.0 avg)
CPU floating-point (%) 0 - 100 (6.9 avg)
10:02:31-10:02:37 (range 6.402s): Mean Memory usage **9.1** M; Mean MPI call duration **5.0** ms; Mean CPU floating-point **6.9** %;

Performance before session 1



14:13:50-14:13:51 (range 1.540s): Mean Memory usage **7.2** M; Mean MPI call duration **0.3** ms; Mean CPU floating-point **15.2** %;

Performance after session 2: 4x speedup

When you can see bottlenecks are at a glance, optimization can be an extremely compelling pastime!

Although this example has been at workstation scale for practical reasons, at no time have we had to look through lists of processes in MAP's GUI. The code navigation and analysis techniques covered apply equally well at 40,000 processes and above - may all your codes scale so well!

Happy profiling,
Mark O'Connor
mark@allinea.com.